AD-A242 274

IMENTATION PAGE

Form Approved
OPM No. 0704-0188

to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data s regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington sports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 29 Nov 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE
R.R. Software, Inc., Janus/Ada 2.2.0 PHAR Lap/DOS, IBM PS/2, MOD. 80 Phar Lap/DOS 3.3 (Host & Target), 901120W11088

5. FUNDING NUMBERS

DTIC ELECTE NOV 5 1991 S C D

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-436-0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
No software available for destribution per Michelle Lee, ADA 11/4/91 Telecon mgm

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
R.R. Software, Inc., Janus/Ada 2.2.0 PHAR Lap/DOS, Wright-Patterson AFB, OH, IBM PS/2, MOD. 80 Phar Lap/DOS 3.3 (Host & Target), ACVC 1.11.

91-15079

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 29 November 1990.

Compiler Name and Version: Janus/Ada 2.2.0 Phar Lap/DOS

Host Computer System: IBM PS/2, MOD. 80 (under Phar Lap/DOS 3.3)

Target Computer System: IBM PS/2, MOD. 80 (under MS DOS 3.3)

Customer Agreement Number: 90-08-02-RRS

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901120W1.11088 is awarded to R.R. Software, Inc. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

for Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

AVF Control Number:AVF-VSR-436-0891
1 August 1991
90-08-02-RRS

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901120W1.11088
R.R. Software, Inc.
Janus/Ada 2.2.0 Phar Lap/DOS
IBM PS/2, MOD. 80 Phar Lap/DOS 3.3 => IBM PS/2 MOD. 80 MS DOS 3.3

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Declaration of Conformance

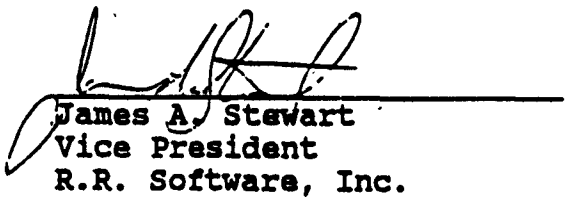
Compiler Implementor : R.R. Software, Inc.
Ada Validation Facility : Wright-Patterson AFB, Ohio 45433-6503
Ada Compiler Validation Capability (ACVC) Version : 1.11

Base Configuration

Ada Compiler Name : Janus/Ada Version : 2.2.0 Phar Lap/DOS
Host Architecture: IBM PS/2, Mod. 80 Host OS & Ver.:Phar Lap/DOS 3.3
Target Architecture: IBM PS/2, Mod.80 Target OS & Ver.: MS DOS 3.3

Implementor's Declaration

I, the undersigned, representing R.R. Software, Inc. have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that R.R. Software, Inc. is the owner of record of the Ada compiler listed above, and as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registration for Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

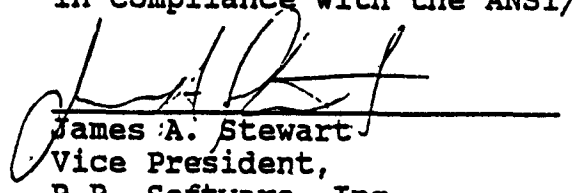


James A. Stewart
Vice President
R.R. Software, Inc.

16 Nov. 1990
Date

Owner's Declaration

I, the undersigned, representing R.R. Software, Inc. take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the ANSI/MIL-STD-1815A.



James A. Stewart
Vice President,
R.R. Software, Inc.

16 Nov. 1990
Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

Reference Manual for the Ada Programming Language, [Ada83]
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Ada Compiler Validation Procedures, Version 2.1, [Pro90]
Ada Joint Program Office, August 1990.

Ada Compiler Validation Capability User's Guide, [UG89] 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 21 November 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 48 or greater.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

D55A03E..H (4 tests) use 31 or more levels of loop nesting which exceeds the capacity of the compiler.

D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

D64005F..G (2 tests) use 10 or more levels of recursive procedure calls nesting which exceeds the capacity of the compiler.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F check for pragma INLINE for procedures and functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A840 use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

AD9004A uses pragma INTERFACE for overloaded subprograms; this implementation rejects this use due to calling conventions. (See section 2.3.)

CDA201C instantiates Unchecked_Conversion with an array type with a non-static index constraint; this implementation does not support Unchecked_Conversion for types with non-static constraints.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	----------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

IMPLEMENTATION DEPENDENCIES

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

EE2201D uses instantiations of package SEQUENTIAL_IO with unconstrained array types; this implementation raises USE_ERROR on the attempt to create a file of such type.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

EE2401D uses instantiations of package DIRECT_IO with unconstrained array types; this implementation raises USE_ERROR on the attempt to create a file of such type.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 88 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002A	B26005A	B27005A
B29001A	B37106A	B51001A	B53003A	B55A01A	B63001A
B63001B	B73004B	B83003B	B83004B	B83004C	B83004D
B83004F	B83030D	B83E01C	B83E01D	B83E01E	B83E01F
C85006A	C85006B	C85006C	C85006D	C85006E	B91001H
BA1001A	BA1001B	BA1001C	BA1010A	BA1010D	BA1101A
BA1101E	BA3006A	BA3006B	BA3007B	BA3008A	BA3008B
BA3013A	BC2001D	BC2001E	BC3005B	BD2B03A	BD2D03A
BD4003A					

C85006A..E (5 tests) were graded passed by Test Modification as directed by the AVO. This implementation generates more object code for these tests than it can contain in a single compilation unit. Each of these tests was split into five equivalent subtests.

The tests below were graded passed by Test Modification as directed by the AVO. These tests all use one of the generic support procedures, Length_Check or Enum_Check (in support files LENCHECK.ADA & ENUMCHEK.ADA), which use the generic procedure Unchecked_Conversion. This implementation rejects instantiations of Unchecked_Conversion with array types that have non-static index ranges. The AVO ruled that since this issue was not addressed by AI-00590, which addresses required support for Unchecked_Conversion, and since AI-00590 is considered not binding under ACVC 1.11, the support procedures could be modified to remove the use of Unchecked_Conversion. Lines 40..43, 50, and 56..58 in LENCHECK and lines 42, 43, and 58..63 in ENUMCHEK were commented out.

CD1009A	CD1009I	CD1009M	CD1009V	CD1009W	CD1C03A
CD1C04D	CD2A21A..C	CD2A22J	CD2A23A..B	CD2A24A	CD2A31A..C
CD2A81A	CD3014C	CD3014F	CD3015C	CD3015E..F	CD3015H
CD3015K	CD3022A	CD4061A			

BD4006A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that non-static values in component and alignment clauses are rejected; but static alignment values of 8, 16, & 32 are assumed to be supported. This implementation supports only values 1 & 2; it rejects the clauses at lines 42, 48, 58, and 63, which are not marked as errors.

AD9001B was graded passed by Processing Modification as directed by the AVO. This test checks that, if pragma INTERFACE is supported, no bodies are required for interfaced subprograms. This implementation requires that some foreign bodies exist, even if the subprograms are not called. This test was processed in an environment in which implementor-supplied foreign bodies were present.

AD9004A was graded inapplicable by Evaluation Modification as directed by the AVO. This test uses a single INTERFACE pragma for several overloaded procedure and function subprograms; this implementation does not support the pragma in such circumstances due to the calling conventions of the interfaced language, and thus rejects the pragma.

CDA201C was graded inapplicable by Evaluation Modification as directed by the AVO. This test instantiates Unchecked_Conversion with an array type with a non-static index constraint; this implementation does not support Unchecked_Conversion for unconstrained types and so rejects the instantiation. The AVO ruled that various restrictions on Unchecked_Conversion may be accepted for validation under ACVC 1.11, because AI-00590, which addresses Unchecked_Conversion, did not show an ARG consensus at the time of ACVC 1.11's release.

CE2108B, CE2108D, and CE3112B were graded passed by Test Modification as directed by the AVO. These tests, respectively, check that temporary files that were created by (earlier-processed) CE2108A, CE2108C, and CE3112A are not accessible after the completion of those tests. However, these tests also create temporary files. This implementation gives the same names to the temporary files in both the earlier- and later-processed tests of each pair; thus, CE2108B, CE2108D, and CE3112B report failed, as though they have accessed the earlier-created files. The tests were modified to remove the code that created the (later) temporary file; these modified tests were passed. Lines 45..64 were commented out in CE2108B and CE2108D; lines 40..48 were commented out in CE3112B.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Isaac Pentinmaki
R.R. Software, Inc.
P.O. Box 1512
Madison, WI 53701

For a point of contact for sales information about this Ada implementation system, see:

Jim Stewart
R.R. Software, Inc.
P.O. Box 1512
Madison, WI 53701

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 Summary Of Test Results

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

PROCESSING INFORMATION

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system — if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3773
b) Total Number of Withdrawn Tests	83
c) Processed Inapplicable Tests	113
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	314
g) Total Number of Tests for ACVC 1.11	4170

3.3 TEST EXECUTION

The diskettes containing the customized test suite (see section 1.3) were taken on-site by the validation team for processing. The contents of the diskettes were installed onto a Northgate 386 with DOS 3.30 and then archived for installation on the actual host computer. The files were restored onto an IBM PS/2 Model 80 with DOS 3.30.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

The options used for Janus/Ada are:

/Q - Quiet error messages - suppresses user prompting on errors. Necessary for running B-Tests; otherwise every error would have to be responded to.

/W - Warnings off - warnings were suppressed mainly because of the many confusing warnings the validation tests produce. Many validation tests have intentional errors (such as an expression which always raises an exception, use of null ranges, unreachable code, etc.). The large volume of warnings produced made it difficult to grade the B-Tests in particular, so they were suppressed.

/BS - Brief Statistics. This was also used to cut the amount of output produced by the compiler during compile time.

/S? - Used this option to re-direct the compiler scratch files into a Ram disk where possible (? is replaced by a drive path), thus speeding up the compiles.

/O1 - Memory model 1 - this directs the compiler to use memory model 1 for the output. This model allows much more code than memory model 0, and is necessary in order to have a few large tests be able to run.

/D - Debugging code off - this directs the compiler to not generate any debugging code (generally line numbers and walkbacks). This was also used to cut the space used by the tests.

All other options used their default values.

Then, all of the non-B-Tests were linked with the options:

/Q - Quiet error messages - suppresses user prompting on errors. Necessary for running L-Tests; otherwise every error would have to be responded to.

/T - Trim unused code - this option directs the linker to remove unused subroutines from the result file. This can make as much as a 30K space saving in the result file.

/B - Brief Statistics. This was also used to cut the amount of output produced by the Linker.

/O1 - Memory model 1 - to match the compiler memory model.

All other options used their default values.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

```
$MAX_STRING_LITERAL   '"' & (1..V-2 => 'A') & '"'
```

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	16
$ALIGNMENT	2
$COUNT_LAST	32_767
$DEFAULT_MEM_SIZE	65536
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MS_DOS2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	(0, 16#40#)
$ENTRY_ADDRESS1	(0, 16#05#)
$ENTRY_ADDRESS2	(0, 16#01#)
$FIELD_LAST	32_767
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	CANNOT_RESTRICT_FILE_CAPACITY
$GREATER_THAN_DURATION	300_000.0
$GREATER_THAN_DURATION_BASE_LAST	1.0E6
$GREATER_THAN_FLOAT_BASE_LAST	1.0E+40
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E308

$HIGH_PRIORITY          0

$ILLEGAL_EXTERNAL_FILE_NAME1
                        /NODIRECTORY/FILENAME

$ILLEGAL_EXTERNAL_FILE_NAME2
                        <BAD/^^>

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006E1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     MASM

$LESS_THAN_DURATION     -305_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -1.0E6

$LINE_TERMINATOR        ASCII.CR & ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -214783648
```

MACRO PARAMETERS

$NAME	NO_SUCH_INTEGER_TYPE
$NAME_LIST	MS_DOS2
$NAME_SPECIFICATION1	D:/VALID/X2120A
$NAME_SPECIFICATION2	D:/VALID/X2120B
$NAME_SPECIFICATION3	D:/VALID/X3119A
$NEG_BASED_INT	16#FFFF_FFFF#
$NEW_MEM_SIZE	65536
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MS_DOS2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	16
$TASK_STORAGE_SIZE	512
$TICK	0.01
$VARIABLE_ADDRESS	FCNDECL.SOME_VAR'ADDRESS
$VARIABLE_ADDRESS1	FCNDECL.SOME_VAR2'ADDRESS
$VARIABLE_ADDRESS2	FCNDECL.SOME_VAR3'ADDRESS
$YOUR_PRAGMA	ALL_CHECKS

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation are provided by the customer and can be found in Appendix F, section F.9, page F-14.

LINKER OPTIONS

The linker options of this Ada implementation are provided by the customer and can be found in Appendix F, section F.9, page F-14.

Jlink Manual

The Janus/Ada linker is used to combine a main program with system and user defined compilation units to create an executable program. Only object code files - that is, JRL (Janus relocatable) and SRL (specification relocatable) files - created by the Janus/Ada compiler, assembler, or other compatible products can be combined. For more information on SRL and JRL files, see Section 10.1 of the compiler manual. The linker only needs to be supplied with the name of the object code file for the main program. It will search disks (including a swap disk, if desired) for the required units. It determines the loading order of the units, and finally produces an executable program. The linker is disk based, so any possible sized program may be linked with JLINK. It may not be possible to produce an easier to use linker.

Using the Linker

The user only needs to type the command

```
JLINK [d:]prog_name
```

to run the linker. The disk name d: specifies the disk on which to look for the main program. The prog_name is the name of the object code file for the main program, *without* the .SRL or .JRL file name extension. The result file is placed into the file with the same name as the main program object code, and the file name extension appropriate for your system (.COM or .EXE for MS-DOS). The result file is placed on the disk where the main program is found.

Examples:

```
JLINK TESTPROG
   -- Links Testprog to make an executable program.
JLINK B:QSORT
   -- Links Qsort (which will be found on the B: disk)
   -- into an executable program
```

This is all of the information needed to use the linker. More details about the operation of the linker will be found on succeeding pages.

Linker Operation

The linker operates automatically. However, to better understand the operation of the linker options (below), a brief description of the linker operation is given here.

The following disk (drive) name definitions are used throughout the rest of the linker manual.

The *default* disk is the one currently logged in on your system. This is the disk name which appears in the system prompt.

The *source* disk is the disk which is specified on the command line as the source of the main program. Unless a different disk is specified on the command line, it will be the default disk.

The *destination* disk is the disk to which the output executable file will be written. Unless a different disk is specified on the command line, it will be the source disk.

The *swap* disk is a disk specified on the command line which allows a whole set of disks to be searched by swapping each disk into a disk drive. Any disk other than the destination disk may be used for this purpose. There is no swap disk unless the swap option is used.

In many cases, the default, source, and destination disks refer to the same hard disk, and the swap disk is not needed.

There are two kinds of segments (memory areas) referred to in this manual. Physical segments are the segments imposed by the underlying hardware. A logical segment is the segmenting imposed by the Janus/Ada compiler and assembler. Each unit is divided into three logical segments - code, data, and constant. The linker's job is to combine many logical segments into the proper number of physical segments. Whenever a segment is referred to in the rest of this manual, it means a logical segment, unless otherwise noted.

The mapping between physical segments and logical segments is specified by the program's *memory model*. Many machines with modern architectures allow the mapping of the many logical segments into one large physical segment. These machines require only one memory model. Other machines (like the 8086 family) have an architecture which makes the mapping more difficult. The different models generate different amounts of code, run at different speeds, and have different limits on the size of programs.

On the 8086 (and chips in the same family), the default memory model will generally cause the linker to produce a COM file. This memory model, called Model 0, has a maximum of 64K of code in a program. To allow the use of more code in a program, one may use the /O1 (Model 1) option in both the compiler and in the linker (see below). This option causes the linker to produce an EXE file; the code size is restricted then only by your operating system and by the amount of memory available on your machine. Note that some versions of the Janus/Ada compiler are sold without the libraries to allow the use of Model 1.

All compilation units in a given program must be compiled with the same memory model, and the program must be linked with that same model; the compiler and the linker enforce this rule.

The linker operates in three phases. The first phase loads the headers of all of the units, and thereby determines which units are needed. This phase also records the disks where the units are found (for later use) and creates a table of entry points (places where other units may access this one). The disks are searched in the following order: first the source disk, then the default disk, and then the swap disk until all of the units are found (or the linker is aborted by the user).

The second phase does not use the disks at all. The table of units to load created by the first phase is used to determine the loading order of the units. This step is necessary due to the elaboration rules of Ada (See Section 10.5 of the compiler manual), since Janus/Ada loads units in the order in which they are to be elaborated. The load address of each unit is determined in this step. The second phase tries to minimize the amount of disk swapping that will be necessary in the third phase.

The third phase reads in each unit, fixes up all external references and relocatable items, and writes out the resulting executable file. [External references are usages of items declared outside of the unit being linked. The linkage information held in an external reference is replaced by actual machine addresses when it is 'fixed up']

If the swap option is not used, as is usually the case when compiling from a hard disk, the fixing up is done without any user intervention. A prompt is issued each time the swap disk needs to be changed, with a list of the units which should be on the disk inserted. The linker tries to be friendly about errors in this process - if the designated file is not found, it just asks again for the disk to be inserted.

Linker Command Line and Options

The linker command line looks like:

```
JLINK [D:]prog_name{/options}
```

The prog_name is the name of the object code file (SRL or JRL file) of the main program, without the ".SRL" or ".JRL" extension. (We will refer to this simply as an SRL file from here on in this manual; however, everything said about the main program's SRL file should be understood to apply if the main program has generated a JRL [as it would if a separate specification was provided]). The optional disk name D allows the source disk to be specified as some other disk than the default disk. The main program's SRL file should be found on the source disk. The main program must be a Janus/Ada parameterless procedure.

Note:
No file name extension is allowed.

In general, the name of the SRL file for the main program corresponds to the first eight letters of the unit name for the main program. If that name had already been used by some other unit, however, then the SRL file name will vary. This can only happen if some other unit in the same program library has the same first eight letters as the main program. For more information on Janus/Ada file naming conventions, see Section 10.1 of the Janus/Ada compiler manual.

Many users will never need any options; most that do will need only the /O1 option to allow for the larger memory model, the /F2 option to use hardware floating point, the /T option to reduce the size of the executable, or occasionally the /E, /L, /Q, or /S options. However, all the following options are provided to increase the flexibility of the linker

Chhhh Set the starting address of the code within the physical code segment to hhhh (hexadecimal). The physical code segment will still be allocated from zero by the operating system. The physical code segment can be loaded anywhere in special applications, so this option will be used very rarely. The resulting program will not work under MS-DOS.

Dhhhh Set the starting address of the data within the physical data segment to hhhh (hexadecimal). The physical data segment will still be allocated from zero. The physical data segment allocation

address can be changed by modifying JLIB86. This option will only be used for very special applications. The resulting program will not work under MS-DOS unless JLIB86 is modified.

E Create an EXE file. This is assumed if the /O1 option is given. This allows allow a somewhat larger total program size if memory model 0 is used, by using different physical segments for the logical constant segment and the logical code segment.

Fn Use the class n (where n is 0 or 2) floating point library. If a floating point library of the specified class already exists in the link, then it is used. Otherwise, the default floating point library is used. If this option is not used, the class 2 library is used if *any* unit was compiled with the /F (hardware floating point) compiler option, or otherwise contains hardware floating point operations; and the class 0 library is used otherwise. The classes are:

0 Universal software floating point (FLOATOPS).

1 Not supported in our Ada compilers.

2 Universal hardware floating point (FLOAT87 on the 8086 series). *Warning*: In some versions of Janus/Ada, the hardware of the 8087 chip can cause some surprising results when using this option. Check Appendix Section L.2 in the compiler manual to see if your version is affected.

3 Not supported in our Ada compilers.

This option allows the use of different forms of floating point support without recompiling the program. It also eliminates the possibility of more than one floating point library being used in a single program (which does not work!). An error is generated if you try to use software support with units compiled with the /F option.

L Turns on the listing option. The current unit being worked on is printed, and a table of addresses for each unit is listed on the console. These can be used for debugging. The format of the table is:

Pack-name Code-hhhh Data-hhhh Const-hhhh

where hhhh is the hex value of the first address assigned to that unit's code, data, or constant segment. The order given for the units in the second and third passes of the linker specifies the

elaboration order of the various units (see Section 10.5 of the compiler manual).

Mhhhh Set the minimum size of the physical code segment plus the physical constant segment. The size hhhh is in paragraphs (16 byte increments). This forces the start of the physical data segment to start hhhh0 (hex) bytes after the start of the physical code segment. If this option is used, and the physical code segment plus the physical constant segment size exceeds the minimum size, a warning message is produced. This option is primarily useful for programs which use the Chain procedure and wish to preserve the physical data segment. The option essentially causes the physical data segment to be in the same absolute locations in each program which uses it.
This option may only be used with the small memory model (model 0).

On Use memory model n (when n is a number between 0 and 7). This option is used to specify an alternative memory model to the small model used by default. All units linked must be compiled or assembled with the specified memory model. The memory models are:

0 Small code. (64K code, 64K data, 64K constants). The default model, and the only one supported in the C-Pak. This model generates a COM file, unless the /E option is specified. A COM file is limited to 64K total for both code and constants. If more is necessary, use the /E option, or model 1.

1 Large code. (Unlimited code, 64K data, 64K constants). The code size is limited by available memory. This model is appropriate for larger programs. This model generates a .EXE file. Programs that use this model will typically be about ten to twenty percent bigger and slower than the equivalent program using model 0.

2-7 Unused.

Q Produce quiet error messages. In the normal mode of operation, the link waits after every error so that the user can see the error without it scrolling off the screen. This option suppresses those prompts.

Rpath — Re-direct the linker output to the specified path. the default is the same path as filename.

Spath — Use the specified path as the swap disk. The linker will abort if all of the libraries needed in the current program are not found on the source or default disk when no swap disk is specified. If a swap disk is specified, a prompt will be issued for the user to change the swap disk. The linking may be aborted at any time during a swap. The swap disk may not be the same as the destination disk. Any other path may be used. This option is most useful on systems with limited disk capacity (i.e. systems with less than 720K per disk).

T — Trim out unreachable subprograms from units that were compiled with the compiler's /T option (see Appendix Section H.1 of the compiler manual); this can significantly reduce the size of the executable file; at the cost of slowing down the linking process. Using this option invokes an extra pass, between the first and second linker passes, to do the requested trimming. We strongly recommend using this option on (at least) final versions of programs.

Uhhhh — Set the starting address of the constants within the physical constant segment to hhhh hexadecimal. The physical constant segment will still be allocated from zero. The physical constant segment allocation address can be changed by modifying JLIB86. This option will only be used for very special applications. The resulting program will not work under MS-DOS unless JLIB86 is modified.

X — Produce an eXtra detailed link map for the program. This will be left in a file called <prog_name>.LNK, where <prog_name> is the name of the main program. This file is mainly for the use of debuggers and other future tools.

Examples:

```
JLINK B:TESTPROG/RD
  -- Link Testprog from the B: disk, and put the
  -- result on the D: disk.
JLINK C:CHAIN/SB/L
  -- Link Chain from the C: disk, search the B: disk
  -- for needed units allowing the user to swap disks
```

```
   -- when needed. Put the result onto the C. disk,
   -- and display a listing of the link addresses.
JLINK B:SAVEDATA/M8000/F2
   -- Link SaveData from the B: disk, forcing the code size
   -- to be at least 8000 Hex bytes. Use floating point
   -- model #2.
```

Error Messages

All errors except warnings are fatal. The linker will prompt the user after an error so that the user is sure to notice the error. (These prompts may be turned off by the /Q option). Fatal errors abort the linker.

Some error messages mention a JRL file explicitly. The meaning may actually be for a SRL file; the error messages only mention JRL for brevity.

Warnings

Some units use Floating Point Hardware, yet a Software unit was specified

> The software and hardware floating point should not be mixed. If you have a floating point co-processor on your target machine, we recommend using only hardware floating point; otherwise, use only software floating point.

The Code segment is larger than the minimum

> (See /M option, above, for explanation)

The unit xxxx is obsolete because yyyy was recompiled
It will be ignored

> The unit xxxx has an optional body, and that body was made obsolete by recompiling yyyy (one of the units withed by xxxx). Ada's rules state that xxxx must be ignored unless it is recompiled. The linker is just letting you know that this happened; if you really want to have the body of xxxx linked in, you should recompile it. This same message can appear as an error if xxxx is anything other than an optional body.

Command Line Errors

These are all caused by an incorrect command line.

Command Line Option Unknown

An option following a slash does not correspond to any legal linker option.

Disk Name too long

The disk name in the command line may have at most one letter.

Extension too long

At most three letters are allowed in a file name extension. (This message comes from our standard command line parser. In fact, the linker does not allow *any* file name extensions)

Extensions not Allowed in Linker

The unit name given to link must not have a file name extension.

File Name too long

The file name in the command line should be at most eight letters long. This is no longer checked in most versions of the linker, so this error should not occur.

Garbage on end of command line

The linker cannot understand some or all of the command line. Make sure that the syntax of the command line matches that listed in the section called "Linker Command Line and Options," above.

Illegal Disk Name for Option

A disk name for the /R or /S option was not in the range A..W.

Illegal Value for Option

The value given with an option that needs one (/F, /O) is illegal or out of range.

Missing Disk Name for Option

An option requiring a disk name (/R, /S) does not have one.

Missing Value for Option

A value was expected following an option (probably /F or /O).

Multiple Colons in File Name

The file name listed in the command line may not have multiple colons.

Multiple Periods in File Name

The file name listed in the command line may not have multiple periods. (This message comes from our standard command line parser. In fact, the linker does not allow file name extensions, so no periods are allowed)

No File Name Present

No file name was found on the command line.

No Hex Number given for option

An option requiring a hexadecimal number (/C, /D, or /M) does not have one.

No option after slash on command line

There was a slash on the command line without anything following it.

Paths not allowed in Linker

The file name in the command line may not include a path.

The Swap disk cannot be the same as the destination disk

Most operating systems do not allow the changing of disks which are being written to; therefore the destination disk cannot be used as the swap disk.

Too many digits in hex number

A hex number specified in the /C, /D, or /M options may only have 4 digits.

Generic Unit Errors

Illegal generic instantiation in xxxx
The instantiation on line number yyyy is indirectly circular.
A unit may not instantiate a unit which instantiates the original unit.

A generic unit cannot instantiate a generic unit which, directly or indirectly, causes an instantiation of the first unit. This error can sometimes be determined at compile time. In the case of separately compiled generics, however, it may not be detected until link time.

In this case, the error message gives the name of the object code file that was produced when one of the offending instantiations was compiled, as well as the line number where that instantiation occurred.

Illegal generic instantiation in xxxx
Parameter number yyyy in the instantiation on line number zzzz
must not be an unconstrained type.

Certain restrictions apply to generic actual types if the corresponding generic formal type is used in certain ways in the body of the generic unit. In particular, the generic actual type may not be either an unconstrained array subtype (see Section 3.6 of the compiler manual) or a discriminated record subtype with no defaults for the discriminants (see Section 3.7.2 of the compiler manual) if the corresponding generic formal type is used in any of the following ways: as the type of a variable declaration; as the type of a component declaration; or as the full declaration of a private type. This restriction prevents using generics to create objects that need to be constrained but are not. This error can sometimes be determined at compile time. In the case of separately compiled generics, however, it may not be detected until link time. In this case, the error message gives the following information: the name of the object code file that was produced when the offending instantiation was compiled; the parameter number in the instantiation; and the line number where the instantiation occurred.

Link Errors

Bad .JRL file - Illegal Data Element

The JRL file or SRL file has a bad data element. Recompile the indicated unit and (if it is a specification) any units that depend on it.

Cannot be a Main Program
A Main Program must be a Parameterless Procedure.

The unit that ends up being the main program must be a non-generic procedure, with no parameters.

Cannot use a .COM file for this program.

The small model program has too much code and constants to fit in a COM file. (MS-DOS puts a 64K limit on the size of COM files; larger ones will not load properly.) You should either use the linker's /E and/or /T options if you are not already doing so; decrease the size of the units (perhaps by using the OPTIMIZE

. pragma); or recompile and relink the entire program with the large memory model (model 1).

Code Segment Overflow - Cannot be larger than 64K

You tried to link a small memory model program (model 0) which has more than 64K of code. You should either use the linker's /T option if you are not already doing so; decrease the size of the units (perhaps by using the OPTIMIZE pragma); or recompile and relink the entire program with the large memory model (model 1).

Compilation Units yet to be loaded -
[Unit List]
are not found on either the source or default disks

The file(s) specified were not found on any of the disks, and are required by this program.

Constant Segment Overflow - Cannot be larger than 64K

You tried to link a program with more than 64K of constants. Janus/Ada only allows 64K of constants, due to the architecture of the 8086.

Data Segment Overflow - Cannot be larger than 64K

You tried to link a program with more than 64K of statically allocated data. The maximum physical segment size is 64K, and the present version of Janus/Ada does not allow more than one normal physical data segment. If you get this error, you must move enough data out of the data segment to allow room both for the statically allocated data and for dynamic data, including local variables and objects allocated on the heap. If the package BIGARRAY was supplied with your compiler, you can use it to move some of your data into a special physical data segment.

Dependency Table Overflow

The linker's internal table that says which units are interdependent has overflowed. If you get this error message, your program is too big for Janus/Ada to handle in one piece. Consider using the Janus/Ada chaining library (CHAINLIB) to break up your program into separate passes (CHAINLIB is not provided with all Janus/Ada packages).

Disk Full

The output disk was full.

Entry Point Table Overflow

The entry point table has overflowed. If you get this error message, your program is too big for Janus/Ada to handle in one piece. Consider using the Janus/Ada chaining library (CHAINLIB) to break up your program into separate passes (CHAINLIB is not provided with all Janus/Ada packages).

Error Number Incorrect

The linker tried to use an unused error message. Please contact us with details.

.JRL file not the same on the second reading

This error can only occur if you used two different swap disks on the first and second reading of a given unit's JRL file or SRL file.

Minimum Code Option can only be used with memory model 0

The /M option is allowed only for small model programs.

Missing External Item

An external item was not found in the entry point table. The unit name in which the item was expected to be found is listed with the error message. The most likely reason for this error is an incorrect compilation order. Another possible reason is that the JRL file or SRL file for the given unit has been damaged. Try recompiling the offending unit. If that fails, then try recompiling the entire program (using CORDER, if you have it, to insure that the order is correct). If both of these fail, please contact RR Software.

Not a .JRL file from the current version of Janus/Ada

The JRL or SRL version number (in the file) is not current. Recompile the indicated unit and (if it is a specification) any units that depend on it.

Not enough RAM

This message indicates that there is not enough available random access memory in your machine to run the linker. Janus/Ada requires 640K of random access memory on your machine. If you think you have enough memory, but you get this message, check whether you are running any memory resident programs; such programs decrease the amount of available memory on your machine. If you have enough memory and no memory resident programs, please contact R.R. Software. This message is usually printed with the name of one of JLINK's units; that information will

help our support staff let you know how much more memory you will need to run the linker.

Obsolete Units found

Some of the units that were to be linked are obsolete: that is, some of the units on which they depend have been recompiled more recently than they have. The offending units were listed before this message came out. The appropriate units and any units that depend on them should be recompiled. If you have CORDER, the Janus/Ada compilation order tool, you may wish to use it to recompile all obsolete units.

The following compilation units could not be loaded due to a mutual dependency (probably caused by Elaborate pragmas)
[Unit List]
The units listed below (if any) may also be mutually dependent.
[Unit List]
The mutual dependency must be removed.

The first list of units shows a cycle of units that cannot be loaded. The last unit in the list is required to be elaborated before the first unit in the list, and each other unit in the list is required to be elaborated before the next unit in the list. Hence, there is no legal elaboration order. The reason that each unit in the list is required to be elaborated before another listed unit is one of the following three reasons: the unit to be loaded first is a specification and other unit is its body; the unit to be loaded first is a specification mentioned in a with clause of the other unit, or the unit to be loaded first is a body mentioned in an ELABORATE pragma of the other unit. The second list of units shows other units that were not loadable; these may depend on the units in the cycle, or there may be another cycle.

In theory, this error can only occur if an ELABORATE pragma is present, since otherwise the compilation order gives a proper elaboration order. If you get this error and you have no ELABORATE pragmas in your program, then one of your SRL or JRL files is probably damaged. Recompile your entire program in a proper order

The unit xxxx is obsolete because yyyy was recompiled

Unit xxxx depends on unit yyyy, but yyyy was compiled more recently than xxxx. Recompile xxxx and any units that depend on it. This same message can appear as a warning if xxxx is an optional body.

Too many Compilation Units in one Program

One program may have only 300 compilation units. If you exceed this limit, your program is too big for Janus/Ada to handle in one piece. Consider using the Janus/Ada chaining library (CHAINLIB) to break up your program into separate passes (CHAINLIB is not provided with all Janus/Ada packages).

Too many deletions for /T option

Your program requires the /T option to remove too many procedures. Reduce your use of unneeded procedures, compile some of your units without the /T compiler option, or do not use the /T option.

Too many externals for /T option

Your program has so many external references that the Janus/Ada linker cannot keep track of all of them for the purposes of the trimming done by the /T option. Reduce the number of such references (possibly by suppressing checks), compile some of your units without the /T compiler option, or do not use the /T option.

Too many generic check records

Your program has more direct and indirect generic instantiations than Janus/Ada can handle; too much memory would be needed to check that the instantiations are all legal. Reduce your usage of generics.

Too many relocations for /T option

The program unit being loaded has so many relocations that the Janus/Ada linker cannot keep track of all of them for the purposes of the trimming done by the /T option. Reduce the number of relocations by compiling the offending unit with the optimizer on, compiling the unit without the /T compiler option, or do not use the /T option.

Too much Code for one JRL File

The maximum amount of code (including constants) in one JRL file or SRL file is 32000 bytes. The JRL file or SRL file is probably damaged. Recompile the indicated unit and (if it is a specification) any units that depend on it.

Unit xxxx has Memory Model different than specified

All units must be compiled with the same memory model as that specified. If this message appears with unit JLIB86, then you are not using the Janus/Ada libraries for the correct model; that is, your MS-DOS search path is not correct.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
   ..........

   type INTEGER is range -32768 .. 32767;

   type LONG_INTEGER is range -21474838648 .. 2147483647;

   type FLOAT is digits 6 range -((2.0 ** 128) - (2.0 ** 104)) ..
                                ((2.0 ** 128) - (2.0 ** 104);

   type LONG_FLOAT is digits 15 range -((2.0 ** 1024) - (2.0 ** 971)) ..
                                      ((2.0 ** 1024) - (2.0 ** 971));

   type DURATION is delta 0.00025 range -((2.0 ** 31) - 1)/4096.0 ..
                                        ((2.0 ** 31) - 1)/4096.0;

   ..........

end STANDARD;
```

F Implementation Dependencies

This appendix specifies certain system-dependant characteristics of the Janus/Ada version 2.2.0 386 to DOS compiler.

F.1 Implementation Dependent Pragmas

In addition to the required Ada pragmas, Janus/Ada also provides several others. Some of these pragmas have a textual range. Such pragmas set some value of importance to the compiler, usually a flag that may be On or Off. The value to be used by the compiler at a given point in a program depends on the parameter of the most recent relevant pragma in the text of the program. For flags, if the parameter is the identifier On, then the flag is on; if the parameter is the identifier Off, then the flag is off; if no such pragma has occurred, then a default value is used.

The range of a pragma - even a pragma that usually has a textual range - may vary if the pragma is not inside a compilation unit. This matters only if you put multiple compilation units in a file. The following rules apply:

1) If a pragma is inside a compilation unit, it affects only that unit.
2) If a pragma is outside a compilation unit, it affects all following compilation units in the compilation.

Certain required Ada pragmas, such as INLINE, would follow different rules; however, as it turns out, Janus/Ada ignores all pragmas that would follow different rules.

The following system-dependent pragmas are defined by Janus/Ada. Unless otherwise stated, they may occur anywhere that a pragma may occur.

ALL_CHECKS Takes one of two identifiers On or Off as its argument, and has a textual range. If the argument is Off, then this pragma causes suppression of arithmetic checking (like pragma ARITHCHECK - see below), range checking (like pragma RANGECHECK - see below), storage error checking, and elaboration checking. If the argument is On, then these checks are all performed as usual. Note that pragma ALL_CHECKS does not affect the status of the DEBUG pragma; for the fastest run time code (and the worst run time checking), both ALL_CHECKS and DEBUG should

be turned Off and the pragma OPTIMIZE (Time) should be used. Note also that ALL_CHECKS does not affect the status of the ENUMTAB pragma. Combining check suppression using the pragma ALL_CHECKS and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, ALL_CHECKS may be combined with the Janus/Ada pragmas ARITHCHECK and RANGECHECK; whichever relevant pragma has occurred most recently will determine whether a given check is performed. ALL_CHECKS is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

ARITHCHECK — Takes one of the two identifiers On or Off as its argument, and has a textual range. Where ARITHCHECK is on, the compiler is permitted to (and generally does) not generate checks for situations where it is permitted to raise NUMERIC_ERROR; these checks include overflow checking and checking for division by zero. Combining check suppression using the pragma ARITHCHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, ARITHCHECK may be combined with the Janus/Ada pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. ARITHCHECK is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

CLEANUP — Takes an integer literal in the range 0 .. 3 as its argument, and has a textual range. Using this pragma allows the Janus/Ada run-time system to be less than meticulous about recovering temporary memory space it uses. This pragma can allow for smaller and faster code, but can be dangerous;

certain constructs can cause memory to be used up very quickly. The smaller the parameter, the more danger is permitted. A value of 3 - the default value -causes the run-time system to be its usual immaculate self. A value of 0 causes no reclamation of temporary space. Values of 1 and 2 allow compromising between "cleanliness" and speed. Using values other than 3 adds some risk of your program running out of memory, especially in loops which contain certain constructs.

DEBUG

Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of line number code and procedure name code. When DEBUG is on, such code is generated. When DEBUG is off, no line number code or procedure names are generated. This information is used by the walkback which is generated after a run-time error (e.g., an unhandled exception). The walkback is still generated when DEBUG is off, but the line numbers will be incorrect, and no subprogram names will be printed. DEBUG's initial state can be set by the command line; if no explicit option is given, then DEBUG is initially on. Turning DEBUG off saves space, but causes the loss of much of Janus/Ada's power in describing run time errors.

Notes:
DEBUG should only be turned off when the program has no errors. The information provided on an error when DEBUG is off is not very useful.

If DEBUG is on at the beginning of a subprogram or package specification, then it must be on at the end of the specification. Conversely, if DEBUG is off at the beginning of such a specification, it must be off at the end. If you want DEBUG to be off for an entire compilation, then you can either put a DEBUG pragma in the context clause of the compilation or you can use the appropriate compiler option.

ENUMTAB

Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of enumeration tables. Enumeration tables are used for the attributes IMAGE, VALUE, and WIDTH, and hence to input and output enumeration values. The tables are

generated when ENUMTAB is on. The state of the ENUMTAB flag is significant only at enumeration type definitions. If this pragma is used to prevent generation of a type's enumeration tables, then using the three mentioned attributes causes an erroneous program, with unpredictable results; furthermore, the type should not be used as a generic actual discrete type, and in particular TEXT_IO.ENUMERATION_IO should not be instantiated for the type. If the enumeration type is not needed for any of these purposes, the tables, which use a lot of space, are unnecessary. ENUMTAB is on by default.

OPTIMIZER Takes one of the identifiers On or Off, or an integer literal, as an argument. This pragma turns optimization on or off, either totally or partially. It has a textual range, except that if the global optimizer is turned on for any part of a compilation unit, then it is on for the entire compilation unit. If the identifier is On or Off, then Janus/Ada's optimizers are turned totally on or totally off, as appropriate. An integer literal as an arguement causes optimization to be turned partially on or off.

The following integer literals are meaningful as an argument to this pragma:

1) Turns check elimination optimizations on.
2) Turns the basic block optimizer on.
3) Turns the global optimizer on. If this is on anywhere in a compilation unit, it will be on everywhere in that unit.
4) Turns peephole optimizations on.
5) Puts the optimizer in 'Space' optimization mode (the default).
6) Puts the optimizer in 'Careful' optimization mode. The can take much longer than 'Quick' optimization, but will find more optimizations.
7) Puts the compiler in 'Fastest alignment' mode. Data objects will be aligned for the fastest performance on the target (unless overridden by rep. clauses). This takes more data space.

51) Turns check elimination optimizations off. Useful for finding uninitialized variables.
52) Turns the basic block optimizer off.

53) Turns the global optimizer off.
54) Turns peephole optimizations off.
55) Puts the optimizer in 'Time' optimization mode.
56) Puts the optimizer in 'Quick' optimization mode. This is faster than 'Careful' optimizations, and often will generate nearly the same code.
57) Put the compiler in 'Smallest alignment' mode. Data is only aligned when required or when the performance penalty is severe. Takes less data space.

Other integer literals will be ignored. In general, this pragma should not be mixed with the OPTIMIZE pragma, since one has a textual arange and the other does not; this can lead to surprising situations. However, the OPTIMIZE pragma may be used inside a compilation unit for which pragma OPTIMIZER(On) has been listed before the start of the compilation unit.

PAGE_LENGTH This pragma takes a single integer literal as its argument. It says that a page break should be added to the listing after each occurrence of the given number of lines. The default page length is 32000, so that no page breaks are generated for most programs. Each page starts with a header that looks like the following:

> Janus/Ada Version 2.2.0 compiling file on date at time

RANGECHECK Takes one of the two identifiers On or Off as its argument, and has a textual range. Where RANGECHECK is off, the compiler is permitted to (and generally does) not generate checks for situations where it is expected to raise CONSTRAINT_ERROR; these checks include null pointer checking, discriminant checking, index checking, array length checking, and range checking. Combining check suppression using the pragma RANGECHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, RANGECHECK may be combined with the Janus/Ada pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. RANGECHECK is on by default. Turning any checks off may cause unpredictable results if execution

would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

SYSLIB — This pragma tells the compiler that the current unit is one of the standard Janus/Ada system libraries. It takes as a parameter an integer literal in the range 1 .. 15; only the values 1 through 4 are currently used. For example, system library number 2 provides floating point support. Do not use this pragma unless you are writing a package to replace one of the standard Janus/Ada system libraries.

VERBOSE — Takes On or Off as its argument, and has a textual range. VERBOSE controls the amount of output on an error. If VERBOSE is on, the two lines preceding the error are printed, with an arrow pointing at the error. If VERBOSE is off, only the line number is printed.

VERBOSE(Off):

```
Line 16 at Position 5
*ERROR* Identifier is not defined
```

VERBOSE(On):

```
  15: if X = 10 then
  16:     Z := 10;
---------^
*ERROR* Identifier is not defined
```

The reason for this option is that an error message with VERBOSE on can take a long time to be generated, especially in a large program. VERBOSE's initial condition can be set by the compiler command line.

Pragma INTERFACE is supported for the language MASM. Pragma INTERFACE_NAME can be used to specify a name other than the Ada one as the name of the MASM function called. INTERFACE_NAME takes two parameters, the Ada subprogram name, and a string representing the MASM name for the function. Pragma INTERFACE_NAME is provided so that convienient Ada names can be

used as appropriate, including operator symbols, and so that foreign language names which are not legal Ada identifiers can be interfaced to. If pragma INTERFACE is used in a program, Jbind must be used to link it, and it must be linked with the Interface run-time.

Several required Ada pragmas may have surprising effects in Janus/Ada. The PRIORITY pragma may only take the value 0, since that is the only value in the range System.Priority. Specifying any OPTIMIZE pragma turns on optimization; otherwise, optimization is only done if specified on the compiler's command line. The SUPPRESS pragma is ignored unless it only has one parameter. Also, the following pragmas are always ignored: CONTROLLED, INLINE, MEMORY_SIZE, SHARED, STORAGE_UNIT, and SYSTEM_NAME. Pragma CONTROLLED is always ignored because Janus/Ada does no automatic garbage collection; thus, the effect of pragma CONTROLLED already applies to all access types. Pragma SHARED is similarly ignored: Janus/Ada's non-preemptive task scheduling gives the appropriate effect to all variables. The pragmas INLINE and SUPPRESS (with two parameters) provide recommendations to the compiler; as Ada allows, the recommendations are ignored. The pragmas MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME all attempt to make changes to constants in the System package; in each case, Janus/Ada allows only one value, so that the pragma is ignored.

F.2 Implementation Dependent Attributes

Janus/Ada does not provide any attributes other than the required Ada attributes.

F.3 Specification of the Package SYSTEM

The package System for Janus/Ada has the following definition.

```
package System is

     -- System package for Janus/Ada

     -- Types to define type Address.
     type Word is range 0 .. 65536;
     for Word'Size use 16;
     type Offset_Type is new Word;
     type Address is record
          Offset : Offset_Type;
          Segment : Word;
     end record;
     Function "+" (Left : Address; Right : Offset_Type) Return
                    Address;
```



```
    Function "+" (Left : Offset_Type; Right : Address) Return
                    Address;
    Function "-" (Left : Address; Right : Offset_Type) Return
                    Address;
    Function "-" (Left, Right : Address) Return Offset_Type;

    type Name is (MS_DOS2);

    System_Name : constant Name := MS_DOS2;

    Storage_Unit : constant := 8;
    Memory_Size : constant := 65536;
         -- Note: The actual memory size of a program is
            -- determined dynamically; this is the maximum
            -- number of bytes in the data segment.

    -- System Dependent Named Numbers:
    Min_Int : constant := -2_147_483_648;
    Max_Int : constant :=  2_147_483_647;
    Max_Digits : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta : constant := 2#1.0#E-31;
         -- equivalently, 4.65661287307739257812 5E-10
    Tick : constant := 0.01; -- Some machines have less
            -- accuracy; for example, the IBM PC actually ticks
            -- about every 0.06 seconds.

    -- Other System Dependent Declarations
    subtype Priority is Integer range 0 .. 0;

    type Byte is range 0 .. 255;
    for Byte'Size use 8;

end System;
```

The type Byte in the System package corresponds to the 8-bit machine byte. The type Word is a 16-bit Unsigned Integer type, corresponding to a machine word.

F.4 Restrictions on Representation Clauses

A length clause that specifies T'SIZE has the following restrictions:

> If T is a discrete type, or a fixed point type, then the size expression can given any value between 1 and 32 bits (subject, of course, to allowing enough bits for every possible value). Signed and unsigned representations are supported.

If T is a floating point type, sizes of 32 and 64 bits are supported (corresponding to Float and Long_Float respectively).

If T is an array or record type, the expression must give enough room to represent all of the components of the type in their object representation. This can be smaller than the default size of the type.

If T is an access type or task type, the expression must give the default size for T.

A length clause that specifies T'STORAGE_SIZE for an access type is supported.

Any integer value can be specified. STORAGE_ERROR will be raised if the value is larger than available memory; no space will be allocated if the value is less than or equal to zero.

A length clause that specifies T'STORAGE_SIZE for a task type T is supported. Any integer value can be specified. Values smaller than 256 will be rounded up to 256 (the minimum T'Storage_Size), as the Ada standard does not allow raising an exception in this case.

A length clause that specifies T'SMALL for a fixed point type must give a value (subject to the Ada restrictions) in the range

```
2.0 ** (-99) .. 2.0 ** 99,
```

inclusive.

An enumeration representation clause for a type T may give any integer values within the range System.Min_Int .. System.Max_Int. If a size length clause is not given for the type, the type's size is determined from the literals given. (If all of the literals fit in a byte, then Byte'Size is used; similarly for Integer and Long_Integer).

The expression in an alignment clause in a record representation clause must equal 1 or 2 (to specify Byte or Word alignment respectively). The alignment value is respected for all object creations unless another representation clause explicitly overrides it. (By placing a component at a non-aligned address, for example).

A component clause may give any desired storage location. The size of the record is adjusted upward if no representation clause

has been given, and more space is needed for the specified storage location to be obeyed.

The range for specifying the bits may specify any values within the following limitations (assuming enough bits are allowed for any value of the subtype):

If the component type is a discrete or fixed point type, any value may be specified for the lower bound. The upper bound must satisfy the equation

```
UB - (LB - (LB Mod System.STORAGE_UNIT_SIZE)) <= 32.
```

If the component type is any other type, the lower bound must satisfy

```
LB Mod System.STORAGE_UNIT_SIZE = 0.
```

The upper bound must be

```
UB := LB + T'Size - 1;
```

Janus/Ada supports address clauses on most objects. Address clauses are not allowed on parameters, generic formal parameters, and renamed objects. The address given for an object address clause may be any legal value of type System.Address. It will be interpreted as an absolute machine address, using the segment part as a selector if in the protected mode. It is the user's responsibility to ensure that the value given makes sense (i.e., points at memory, does not overlay other objects, etc.) No other address clauses are supported.

F.5 Implementation Defined Names

Janus/Ada uses no implementation generated names.

F.6 Address Clause Expressions

The address given for an object address clause may be any legal value of type System.Address. It will be interpreted as an absolute machine address, using the segment part as a selector if in the protected mode. It is the user's responsibility to ensure that the value given makes sense (i.e., points at memory, does not overlay other objects, etc.)

F.7 Unchecked_Conversion Restrictions

We first make the following definitions:

A type or subtype is said to be a simple type or a simple subtype (respectively) if it is a scalar (sub)type, an access (sub)type, a task (sub)type, or if it satisfies the following two conditions:

1) If it is an array type or subtype, then it is constrained and its index constraint is static; and
2) If it is a composite type or subtype, then all of its subcomponents have a simple subtype.

A (sub)type which does not meet these conditions is called non-simple. Discriminated records can be simple; variant records can be simple. However, constraints which depend on discriminants are non-simple (because they are non-static).

Janus/Ada imposes the following restriction on instantiations of Unchecked_Conversion: for such an instantiation to be legal, both the source actual subtype and the target actual subtype must be simple subtypes, and they must have the same size.

F.8 Implementation Dependencies of I/O

The syntax of an external file name depends on the operating system being used. Some external files do not really specify disk files; these are called devices. Devices are specified by special file names, and are treated specially by some of the I/O routines.

The syntax of an MS-DOS 2.xx or 3.xx filename is:

[d:][path]filename[.ext]

where "d:" is an optional disk name; "path" is an optional path consisting of directory names, each followed by a backslash; "filename" is the filename (maximum 8 characters); and ".ext" is the extension (or file type). See your MS-DOS manual for a complete description. In addition, the following special device names are recognized

STI: MS-DOS standard input. The same as Standard_Input. Input is buffered by lines, and all MS-DOS line editing characters may be used. Can only be read.
STO: MS-DOS standard output. The same as Standard_Output. Can only be written.
ERR: MS-DOS standard error. The output to this device cannot be redirected. Can only be written.
CON: The console device. Single character input with echoing. Due to the design of MS-DOS, this device can be redirected. Can be read and written.

AUX: The auxiliary device. Can be read or written.
LST: The list (printer) device. Can only be written.
KBD: The console input device. No character interpretation is performed, and there is no character echo. Again, the input to this device can be redirected, so it does not always refer to the physical keyboard.

The MS-DOS device files may also be used (CON, AUX, and PRN without colons ':'). For compatibility reasons, we do not recommend the use of these names.

The MS-DOS 2.xx version of the I/O system will do a search of the default search path (set by the DOS PATH command) if the following conditions are met:

1) No disk name or path is present in the file name; and

2) The name is not that of a device.

Alternatively, you may think of the search being done if the file name does not contain any of the characters ':', '/', or '\'.

The default search path cannot be changed while the program is running, as the path is copied by the Janus/Ada program when it starts running.

Note:
Creates will never cause a path search as they must work in the current directory.

Upon normal completion of a program, any open external files are closed. Nevertheless, to provide portability, we recommend explicitly closing any files that are used.

Sharing external files between multiple file objects causes the corresponding external file to be opened multiple times by the operating system. The effects of this are defined by your operating system. This external file sharing is only allowed if all internal files associated with a single external file are opened only for reading (mode In_File), and no internal file is Created. Use_Error is raised if these requirements are violated. A Reset to a writing mode of a file already opened for reading also raise Use_Error if the external file also is shared by another internal file.

Binary I/O of values of access types will give meaningless results and should not be done. Binary I/O of types which are

not simple types (see definition in Section F.7, above) will raise Use_Error when the file is opened. Such types require specification of the block size in the form, a capability which is not yet supported.

The form parameter for Sequential_IO and Direct_IO is always expected to be the null string.

The type Count in the generic package Direct_IO is defined to have the range 0 .. 2_147_483_647.

Ada specifies the existence of special markers called terminators in a text file. Janus/Ada defines the line terminator to be <LF> (line feed), with or without an additional <CR> (carriage return). The page terminator is the <FF> (form feed) character; if it is not preceded by a <LF>, a line terminator is also assumed.

The file terminator is the end-of-file returned by the host operating system. If no line and/or page terminator directly precedes the file terminator, they are assumed. If the form "Z" is used, the <Ctrl>-Z character also represents the end-of-file. This form is not necessary to correctly read files produced with Janus/Ada and most other programs, but may be occasionally necessary. The only legal forms for text files are "" (the null string) and "Z". All other forms raise USE_ERROR.

If the form is "", the <Ctrl>-Z character is ignored on input. The <CR> character is always ignored on input. (They will not be returned by Get, for instance). All other control characters are sent directly to the user. Output of control characters does not affect the layout that Text_IO generates. In particular, output of a <LF> before a New_Page does not suppress the New_Line caused by the New_Page.

On output, the "Z" form causes the end-of-file to be marked by a <Ctrl>-Z; otherwise, no explicit end-of-file character is used. The character pair <CR> <LF> is written to represent the line terminator. Because <CR> is ignored on input, this is compatible with input.

The type Text_IO.Count has the range 0 .. 32767; the type Text_IO.Field also has the range 0 .. 32767.

IO_Exceptions.USE_ERROR is raised if something cannot be done because of the external file system; such situations arise when one attempts:

- to create or open an external file for writing when the external file is already open (via a different internal file).
- to create or open an external file when the external file is already open for writing (via a different internal file).
- to reset a file to a writing mode when the external file is already open (via a different internal file).
- to write to a full disk (Write, Close);
- to create a file in a full directory (Create);
- to have more files open than the OS allows (Open, Create);
- to open a device with an illegal mode;
- to create, reset, or delete a device;
- to create a file where a protected file (i.e., a directory or read-only file) already exists;
- to delete a protected file;
- to use an illegal form (Open, Create); or
- to open a file for a non-simple type without specifying the block size;
- to open a device for direct I/O.

IO_Exceptions.DEVICE_ERROR is raised if a hardware error other than those covered by USE_ERROR occurs. These situations should never occur, but may on rare occasions. For example, DEVICE_ERROR is raised when:

- a file is not found in a Close or a Delete;
- a seek error occurs on a direct Read or Write; or
- a seek error occurs on a sequential End_Of_File.

The subtypes Standard.Positive and Standard.Natural, used by some I/O routines, have the maximum value 32767.

No package Low_Level_IO is provided.

F.9 Running the compiler and linker

The Janus/Ada compiler is invoked using the following format:

```
JANUS [path] filename [.ext] (/option)
```

where filename is an MS/DOS file name with optional path [path] (here path includes disk names), optional extension [.ext], and compiler options (/option). If no path is specified, the current disk and path is assumed. If no extension is specified, .PKG is assumed.

The compiler options are:

B — Brief error messages. The line in error is not printed (equivalent to turning off pragma VERBOSE).

BS — Brief statistics. Few compiler statistics are printed.

D — Don't generate debugging code (equivalent to turning off pragma DEBUG)

F — Use in-line 8087 instructions for Floating point operations. By default the compiler generates library calls for floating point operations. The 8087 may be used to execute the library calls. A floating point support library is still required, even though this option is used.

L — Create a listing file with name filename.PRN on the same disk as filename. The listing file will be a listing of only the last compilation unit in a file.

Lpath — Create a listing file on specified path 'path'.

Ox — Object code memory model. X is 0 or 1. Memory model 0 creates faster, smaller code, but limits all code in all units of a program to one MS-DOS segment (i.e., 64 kilobytes); Memory model 1 allows code size limited only by your machine and operating system. See the linker (JLINK) manual for more information. Memory model 0 is assumed if this option is not given. The compiler records the memory model for which each library unit was compiled, and it will complain if any mismatches occur. Thus, the compiler enforces that if it is run using the /o1 option, then all of the withed units must have been compiled with the same option.

Q — Quiet error messages. This option causes the compiler not to wait for the user to interact after an error. In the usual mode, the compiler will prompt the user after each error to ask if the compilation should be aborted. This option is useful if the user wants to take a coffee break while the compiler is working, since all user prompts are suppressed. The errors (if any) will not stay on the screen when this option is used; therefore, the console traffic should be sent to the printer or to a file. Be warned that certain syntax errors can cause the compiler to print many error messages for each and every line in the program. A lot of paper could be used this way! Note that the /Q

option disallows disk swapping, even if the /S option is given.

Rpath — Route the SYM, SRL, and JRL files produced by the compiler to the specified path 'path'. The default is the same path as filename.

Spath — Route Scratch files to specified path. This option is useful if you have a RAM disk or if your disk does not have much free space. The use of this option also allows disk swapping to load package specification (.SYM) files. Normally, after both the compiler and source file disks are searched for .SYM files, an error is produced if they are not all found. However, when the /S option is used, the compiler disk may be removed and replaced by a disk to search. The linker has a similar option, which allows the development of large programs on systems with a small disk capacity. Note that disk swapping is not enabled by the /S option if the /Q (quiet option) is also given. The /Q option is intended for batch mode compiles, and its purpose conflicts with the disk swapping. The main problem is that when the /S option is used to put scratch files on a RAM disk, a batch file may stop waiting for a missing .SYM or ERROR.MSG file; such behavior would not be appropriate when /Q is specified.

T — Generate information which allows trimming unused subprograms from the code. This option tells the compiler to generate information which can be used by the remove subprograms from the final code. This option increases the size of the .JRL files produced. We recommend that it be used on reusable libraries of code (like trig. libraries or stack packages) - that is those compilations for which it is likely that some subprograms are not called.

W — Don't print any warning messages. For more control of warning messages, use the following option form (Wx).

Wx — Print only warnings of level less than the specified digit 'x'. The given value of x may be from 1 to 9. The more warnings you are willing to see, the higher the number you should give.

X — Handle eXtra symbol table information. This option is for the use of the JScope debugger and other tools. This option requires large quantities of memory and disk space, and thus should be avoided if possible.

Z Turn on optimization. This has the same effect as if the pragma OPTIMIZE were set to SPACE throughout your compilation.

The default values for the command line options are:

B Error messages are verbose.
BS Statistics are verbose.
D Debug code is generated.
F Library calls are generated for floating point operations.
L No listing file is generated.
O Memory model 0 is used.
Q The compiler prompts for abort after every error.
R The SYM, SRL, and JRL files is put on the same path as the input file.
S Scratch files are put in the current directory.
T No trimming code is produced.
W All warnings are printed.
X Extra symbol table information is not generated.
Z Optimization is done only where so specified by pragmas.

Leading spaces are disregarded between the filename and the call to JANUS. Spaces are otherwise not recommended on the command line. The presence of blanks to separate the options or between the filename and the extension will be ignored.

Examples:

```
JANUS test/Q/L
JANUS test.run/W4
JANUS test
JANUS test   .run /B /W/L
```

The compiler produces a SYM (SYMbol table information) file when a specification is compiled, and a SRL or JRL (Specification ReLocatable or Janus ReLocatable) file when a body is compiled. To make an executable program, the appropriate SRL and JRL files must be linked (combined) with the run-time libraries. This is accomplished by running the Janus/Ada linker, JLINK.

The Janus/Ada linker is invoked using the following format:

```
JLINK [path] filename (/option)
```

Here "filename" is the name of the SRL or JRL file created when the main program was compiled (without the .SRL or .JRL extension) with optional path name [path] (again, the disk name is consider part of the path here), and compiler options (/option). The filename usually corresponds to the first eight

letters of the name of your main program. A path may be specified where the files are to be found. See the linker manual for more detailed directions. We summarize here, however, a few of the most commonly used linking options:

E Create an EXE file. This is assumed if the /O1 option is given. This allows allow a slightly larger total program size if memory model is used.
F0 Use software floating point (the default).
F2 Use hardware (8087) floating point.
L Display lots of information about the loading process.
O0 Use memory model 0 (the default); see the description of the /O option in the compiler, above.
O1 Use memory model 1.
Q Use quiet error messages; i.e., don't wait for the user to interact after an error.
B Use brief statistics.
T Trim unused subprograms from the code. This option tells the linker to remove subprograms which are never called from the final output file. This option reduces space usage of the final file by as much as 30K.

Examples:

```
JLINK test
JLINK test /Q/L
JLINK test/O1/L/F2
```

Note that if you do not have a hardware floating point chip, and if you are using memory model 0, then you generally will not need to use any linker options.